Lesson 7

A Review of Data Structures:

Fully Developing the Container of Knowledge

Wholeness of the Lesson

An analysis of the average-case and worst-case running times of many familiar data structures (for instance, array lists, linked lists, stacks, queues, hashtables, binary search trees) highlights their strengths and potential weaknesses; clarifies which data structures should be used for different purposes; and points to aspects of their performance that could potentially be improved. Likewise, finer levels of intelligence are more expanded but at the same time more discriminating. For this reason, action that arises from a higher level of consciousness spontaneously computes the best path for success and fulfillment.

Overview of the Lesson

- The List Data Type
- Stacks
- Queues
- Hashtables
- Binary Search Trees
- Balanced Binary Search Trees

The List Data Type
- List Types with Running Times

Operation	**ArrayList Running Time**	**LinkedList Running Time**
get(index)		
find(object)		
addFirst(object)		
addLast(object)		
insert(index, object)		
remove(index)		
remove(object)		
Print		

The List Data Type
- List Types with Running Times

Operation	**ArrayList Running Time**	**LinkedList Running Time**
get(index)	O(1)	O(n) (avg and worst)
find(object)	O(n)	O(n)
addFirst(object)	O(n) (restructure needed)	O(1)
addLast(object)	O(1) (unless resize needed)	O(1) (if last pointer maintained)
insert(index, object)	O(n) (restructure needed)	O(n) (find insertion point)
remove(index)	O(n) (restructure needed)	O(n) (find insertion point; doubly linked preferable)
remove(object)	O(n) (find & restructure)	O(n) (find insertion point)
Print	O(n)	O(n)

The Stack ADT

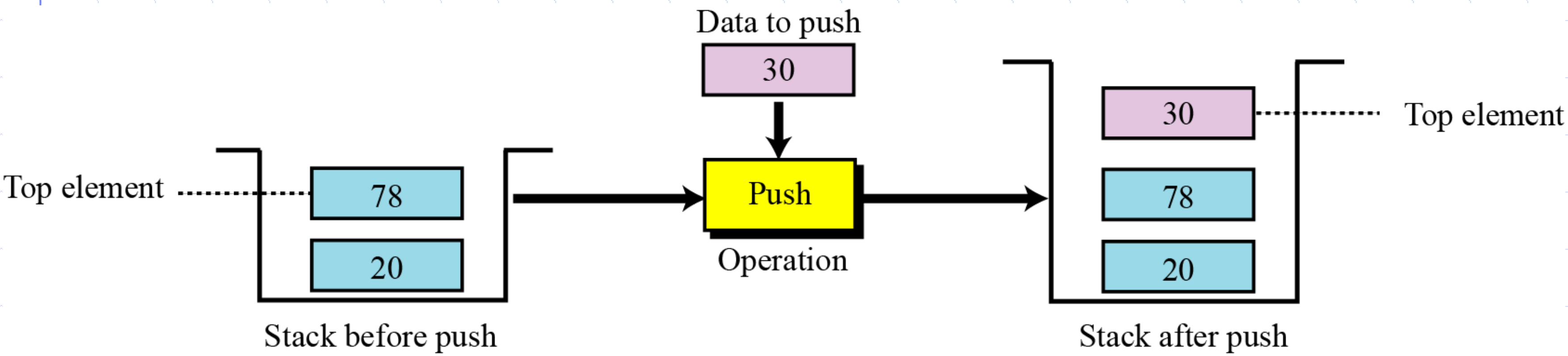
- **Definition:** A STACK is a LIST in which insertions and deletions can occur relative to just one designated position (called the *top of the stack*).

- **Operations:**

pop	remove top of the stack and return this object)
push	insert object as new top of stack
peek	view object at top of the stack without removing it

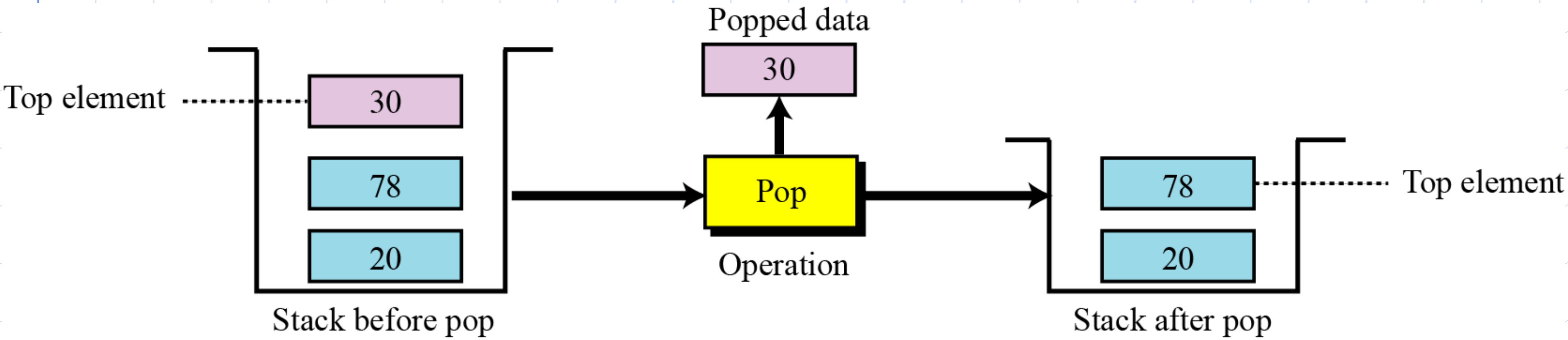
The STACK ADT

- `push` **operation:**



The STACK ADT

- pop operation:



The Stack ADT

- Running Times:
  - All operations, under a reasonable implementation, run in constant time – that is, O(1).
- Implementation:
  - Stacks can be implemented using arrays (rightmost element is top) or linked lists.

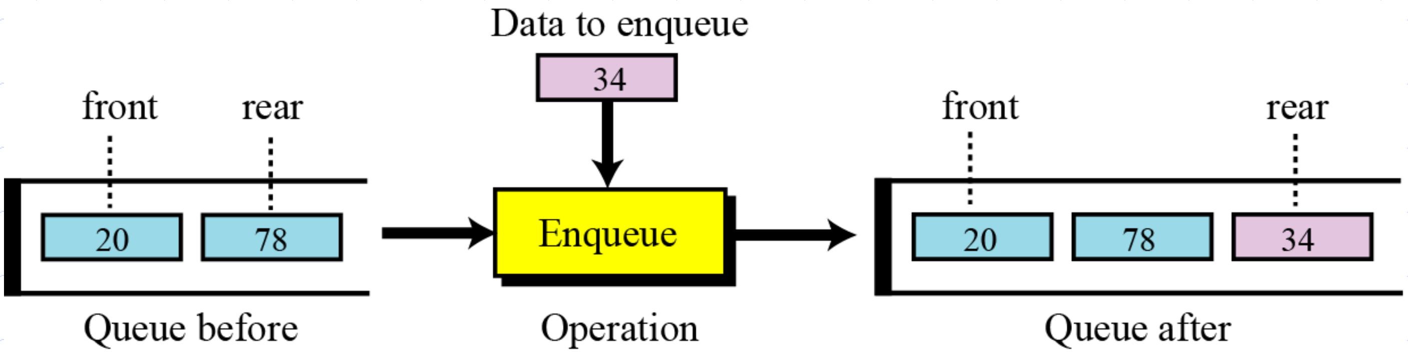
The QUEUE ADT

- **Definition.** Like a STACK, a QUEUE is a specialized LIST in which insertions may occur only at a designated position (the *back*) and deletions may occur only at a designated position (the *front*).
- **Operations**:

dequeue	remove the element at the front (usually also returns this object)
enqueue	insert object at the back
peek	view object at front of queue without removing it

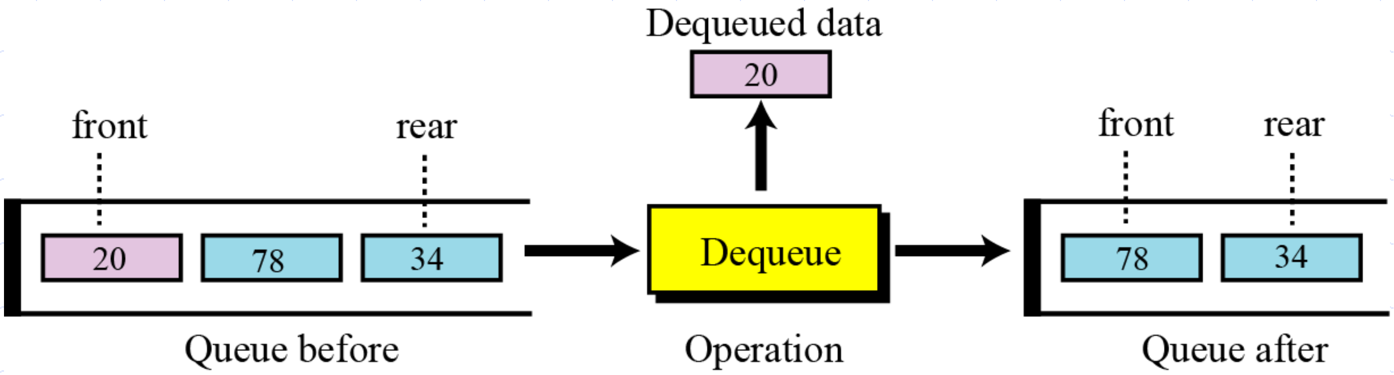
The QUEUE ADT

- `enqueue` **operation:**



The QUEUE ADT

- `dequeue` **operation:**



The QUEUE ADT

- Running Times:
  - All operations, under a reasonable implementation, run in constant time.
- Implementation:
  - Queues can be implemented using an array or linked list.

Main Point

- Stacks and queues achieve O(1) performance of their main operations, which involve either reading/removing the top element or inserting a new element either at the top or the end. Stacks and queues achieve their high level of efficiency by concentrating on a single point of input (top of stack or end of queue) and a single point of output (top of stack or front of queue).
- Stacks and queues make use of the principle from Maharishi Vedic Science that the dynamism of creation arises in the concentration of dynamic intelligence to a point value ("collapse of infinity to a point”).

The Hashtable ADT

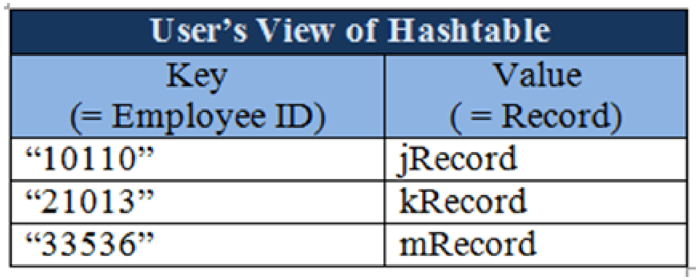
- A Hashtable is a generalization of an array in which any object can be used as a key instead of just integers. A Hashtable has *keys* which are used to look up corresponding *values.*
- Operations:

get	returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.
put	maps the specified key to the specified value in this hashtable.
remove	removes the key (and its corresponding value) from this hashtable.

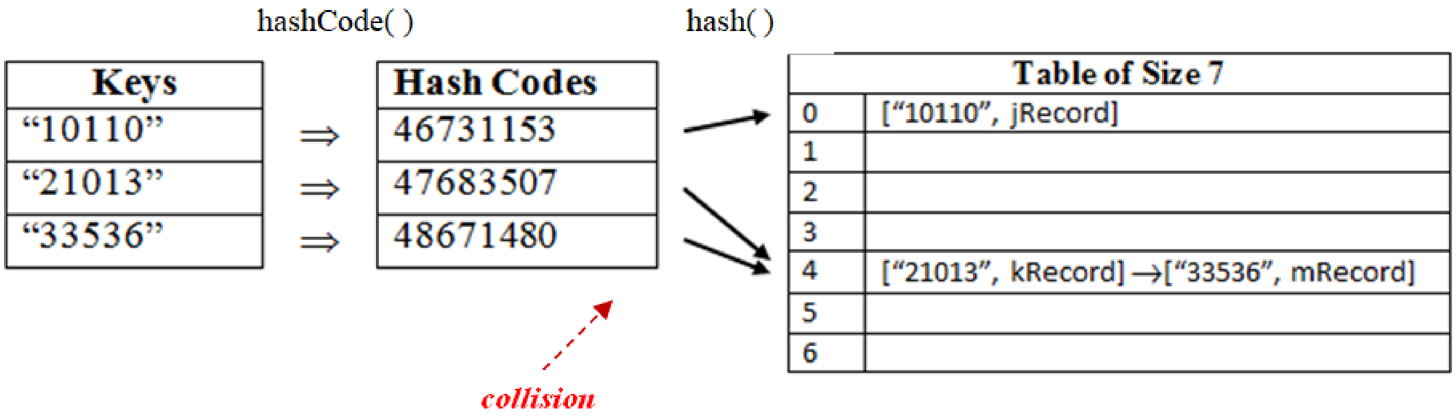
Hashing Strategy

- In a Hashtable, values lying in a large range are compressed into a (typically) much smaller range consisting of array indices.
- This transformation is accomplished in two Steps:
  - create a *hashcode* for each key – typically, this is a big integer
  - create a *hash value* for each hashcode – the hash value is an index in the underlying table for the Hashtable.

Hashing Strategy



User's View of Hashtable	
Key (= Employee ID)	Value (= Record)
"10110"	jRecord
"21013"	kRecord
"33536"	mRecord



Keys
"10110"
"21013"
"33536"

Hash Codes
46731153
47683507
48671480

Table of Size 7	
0	["10110", jRecord]
1	
2	
3	
4	["21013", kRecord] →["33536", mRecord]
5	
6	

$$\text{hashcode}(\text{key}) = \mathit{key.hashCode}() \qquad \text{hash}(\text{hashcode}) = \text{hashcode} \,\%\, \text{tablesize}$$

Running Times for HashMap

- There are several factors that contribute to the running time of hashtable operations:
  - Cost of computing hashCode().
  - Cost of computing hash values.
  - The degree to which these two computations minimize collisions

Cost of Computing hashCode()

- The hashCode method that is used for classes in the Java libraries, when the choice of keys is reasonable, runs in O(1). Here are some examples of "reasonable" keys and how hashCode is computed for them. Suppose f is such an instance variable.
  - If f is Boolean, compute (fvalue ? 1 : 0) (where fvalue is f.booleanValue())
  - If f is a Byte, Character, Short, or Integer, compute (int) fvalue.
  - If f is a Long, compute (int) (fvalue ^ (fvalue >>> 32))
  - If f is a Float, compute Float.floatToIntBits(fvalue)
  - If f is a Double, compute Double.doubleToLongBits(fvalue) which produces a long f1, then return (int) (f1 ^ (f1 >>> 32))

Cost of Computing hashCode()

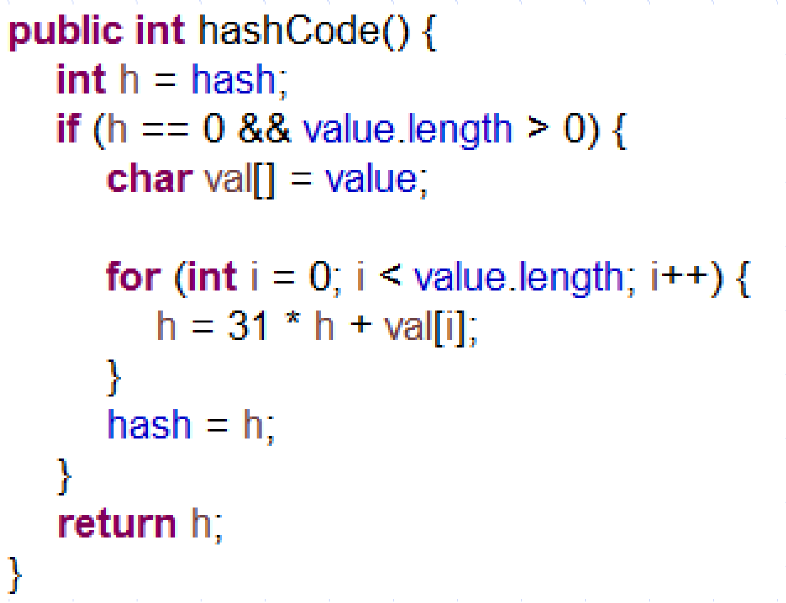
- *Hashing Objects.* Good hashCode functions will create a hashCode based on the instance variables of the object. Suppose your class has instance variables u, v, w and corresponding hashCodes hash_u, hash_v, hash_w (if u, v or w is a primitive hash_u, hash_v, hash_w would be obtained as described in the last slide; otherwise if, say, u is an object of some kind, hash_u = u.hashCode()). Then:

```
@Override
public int hashCode() {
    int result = 17;
    result += 31 * result + hash_u;
    result += 31 * result + hash_v;
    result += 31 * result + hash_w;
    return result;
 }
```

If computing each of the hashCodes required only O(1) time, then running time to execute hashCode() for an instance of the object that has u, v, w as its instance variables is also O(1).

Cost of Computing hashCode()

- *Hashing Strings*. Here is how `hashCode` is overridden in the Java `String` class: (in the `String` class, the variable value stores `this.toCharArray()`).



```
public int hashCode() {
    int h = hash;
    if (h == 0 && value.length > 0) {
        char val[] = value;

        for (int i = 0; i < value.length; i++) {
            h = 31 * h + val[i];
        }
        hash = h;
    }
    return h;
}
```

If all strings used as keys in a hashtable have length < m (for some fixed integer m), then the cost of computing each hashCode is O(m). This should be viewed as a fixed constant running time, therefore O(1).

Cost of Computing hashCode()

- *Cases in which hashCode is expensive.* Whenever keys for a hashtable have variable length without some clear upper bound, the cost of performing hashCode may end up being proportional to (or grow even faster than) the table size itself. In such cases, we cannot claim hashCode runs in O(1) time.

  Example: You could use Java Lists as keys in a hashtable. If you know for sure that the Lists that will be used as keys always have size less than a fixed number m, then hashCodes of these lists can still be computed in O(1) time. But if the Lists may end up containing arbitrarily many elements, then this is no longer the case.

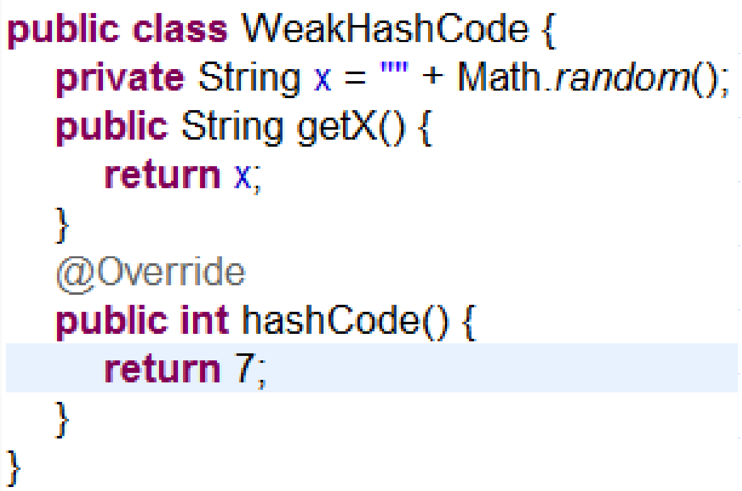
Cost of Computing Hash Values

- *Traditional computation.* Suppose you have an underlying table of size n, so that keys for your hashtable need to be mapped into one of the slots 0, 1, 2, . . ., n – 1. A generally effective method to compute hash values from hash codes is:

```
hashCode  mod  n
```

Cost of Handling Collisions

- When the hash value function *h* maps two *hashCodes* to the same slot, we have a *collision.* When there are too many collisions, the hash table is said to be *skewed.* In the worst case, all *hashCodes* are mapped to the same slot. In that case, all values are stacked up (typically in a list) in a single slot. This results in Θ(*n*) performance for *find, insert,* and *remove* operations.
- **Example**: Here, for typical hashtable implementations, all instances of `WeakHashCode` that are used as keys in a hashtable will be stacked up in slot #7 of the underlying hashtable table.



```
public class WeakHashCode {
    private String x = "" + Math.random();
    public String getX() {
        return x;
    }
    @Override
    public int hashCode() {
        return 7;
    }
}
```

Cost of Handling Collisions

- In order for hashtable operations to be O(1), collisions have to be handled well.
- *Separate Chaining.* Separate chaining provides a way to handle collisions by placing in a list all key/value pairs for which the hash values associated with the key are equal: All key/value pairs belonging to the same slot are lined up in a list that lives in that slot.

Cost of Handling Collisions

- *Load Factor.* Suppose the underlying table of a hashtable has *n* slots and we insert *m* elements. Then the *load factor* for the hashtable is given by

  $$\alpha = m/n.$$

  - Assuming input data has no patterns that would cause the table to be skewed, and assuming hashing distributes keys reasonably evenly, then

    The expected size of the list in each slot in the table is α.
  - Under these assumptions, the running time of put, get, remove, and containsKey is always O(α).
    - put: O(1) to locate the slot (via hashing), O(α) to put the key in the list
    - get: O(1) to locate the slot (via hashing), O(α) to find the requested key in the list
    - remove: O(1) to locate the slot, O(α) to find the requested key in the list, O(1) to remove from the list (recall the list is a *linked* list)
    - containsKey: O(1) to locate the slot (via hashing), O(α) to find the requested key in the list

Cost of Handling Collisions

- When *m* is O(*n*) (that is, number of elements is not "too much" larger than tableSize), then α is O(1).

- *Rehashing.* If the number *m* of elements becomes significantly larger than the table size *n,* then the hashtable is no longer efficient. Java handles this by rehashing the hashtable whenever the load factor gets too big (the threshhold can be set by the user but defaults to 0.75).

Summary on Hashtables

- It is accurate to assert that the hashtable operations get, put, remove, containsKey perform in O(1) time on average, under the following assumptions:
  - Both `hashCode` and `hash` functions perform in O(1). This will be the case if keys are typical (numeric objects, Strings of bounded length, objects whose instance variables are not too complex)
  - The number of elements in the table is O(*n*), where *n* is the table size

- These assumptions are realistic as long as:
  - The developer overrides `hashCode()` in a way that mixes up values adequately
  - The keys used in a hashtable are not too complex (avoiding using potentially long collection classes or strings whose lengths are not uniformly bounded)

Main Point

Hashtables are a generalization of the concept of an array. They support (nearly) random access of table elements by looking up with a (possibly) non-integer key, and therefore their main operations have an average-case running time of O(1). Hashtables illustrate the principle of *Do less and accomplish more* by providing extremely fast implementation of the main List operations.

Binary Search Trees

- Question: What is the best data structure to keep data in sorted order in memory?
- How hard is it to implement this strategy: maintain sorted order to optimize searches ?
  - If we are using an ArrayList, then maintaining the list in sorted order is expensive because each time we add a new element, it must be inserted into the correct spot, and this requires array copy routines.
  - If we are using a LinkedList, it is easy to maintain sorted order since insertions are efficient, but searches are not very efficient. Exercise: What is the running time to carry out BinarySearch in a LinkedList?
- *The Need.* We need a data structure that performs insertions efficiently in order to maintain sorted order (like a linked list) but that also performs finds efficiently (as in an array list where binary search is highly efficient)

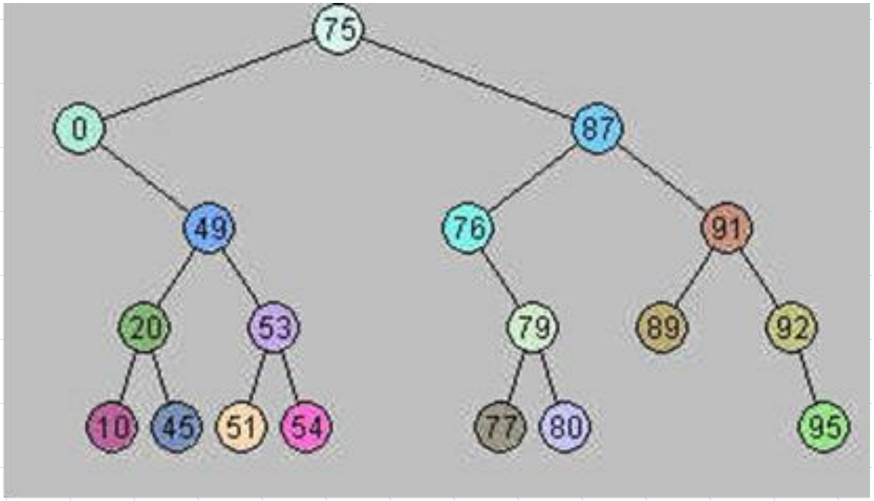
A Solution: Binary Search Trees

- A *binary tree* is a tree that each node has a reference to a left and right child node (though some references may be null).
- A *binary search tree* (BST) is a binary tree in which the BST Rule is satisfied:

<u>BST Rule:</u>

At each node N, every value in the left subtree of N is less than the value at N, and every value in the right subtree of N is greater than the value at N.

For the moment, we assume all values are of type Integer.



A Solution: Binary Search Trees

- The fundamental operations on a BST are:

```
public boolean find(Integer val)
public void insert(Integer val)
public boolean remove(Integer val)
public void print()
```

- When implemented properly, BSTs perform insertions and deletions faster than can be done on Linked Lists and performs any `find` with as much efficiency as BinarySearch on a sorted array.

- In addition, because of the BST Rule, the BST keeps all data in sorted order, and the algorithm for displaying all data in its sorted order is very efficient.

Insertion into a BST

Recursive algorithm for insertion of Integer x [an iterative implementation is given in the demo code]

- If the root of the tree being examined is null, create a new root having value x
- Else:
  - if x is less than the value in the root, insert to the left subtree of the root
  - if x is bigger than the value in the root, insert to the right subtree

Exercise: Insert the following into an initially empty BST: 1,8,2,3,9,5,4

Searching a BST

Recursive algorithm for finding an Integer x

The recursive *find* operation for BSTs is in reality the binary search algorithm in the context of BSTs.

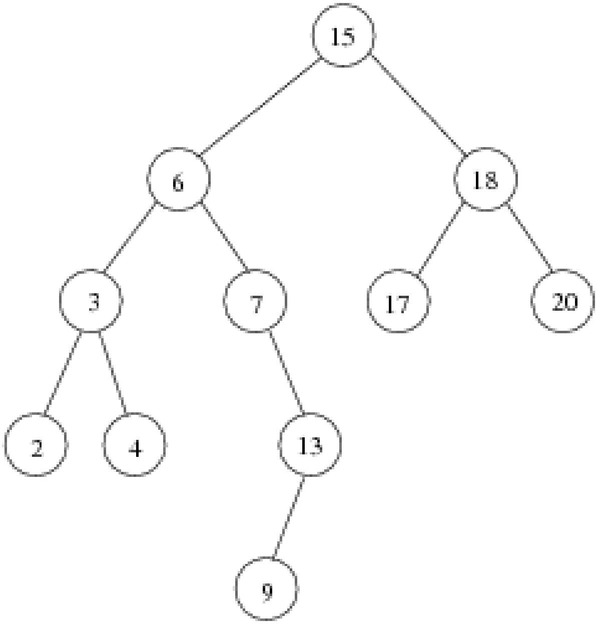
1. If the root is null, return false

2. If the value in the root equals x, return true

3. If x is less than the value in the root, return the result of searching the left subtree

4. If x is greater than the value in the root,
 return the result of searching
 the right subtree

Example: search for 13



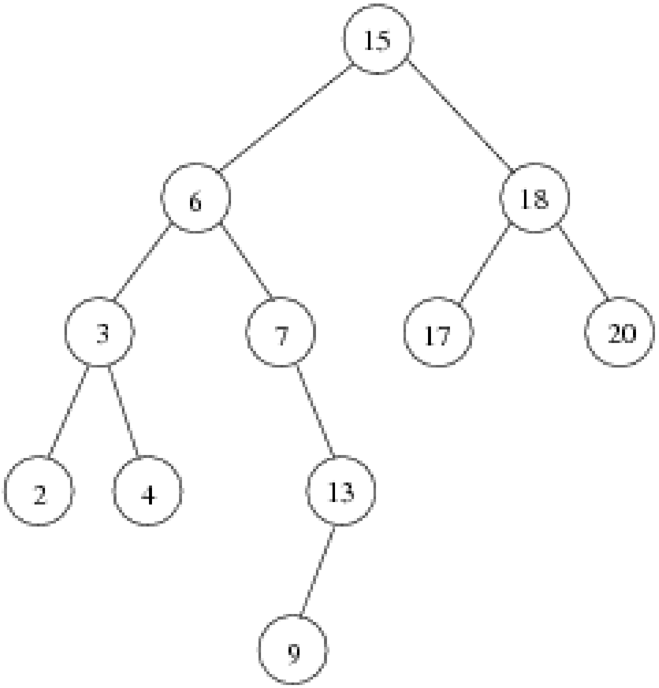
BST In-Order Traversal

Recursive print algorithm – outputs values in every node in sorted order

1. If the root is null, return
2. Print the left subtree of the root, in sorted order
3. Print the root
4. Print the right subtree of the root, in sorted order

BST In-Order Traversal

- Example:

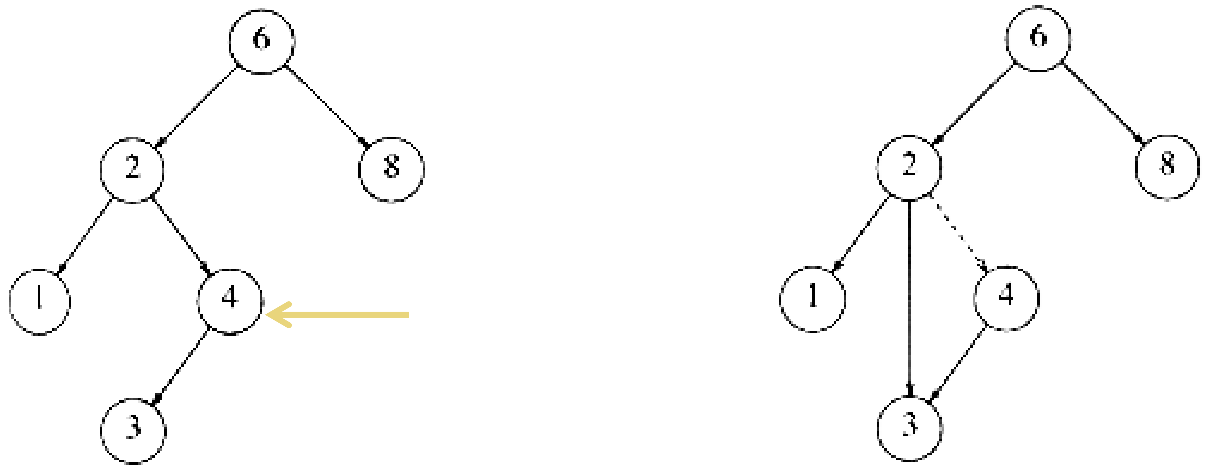


In-Order: 2, 3, 4, 6, 7, 9, 13, 15, 17, 18, 20

Deletions in a BST

Algorithm to remove Integer x

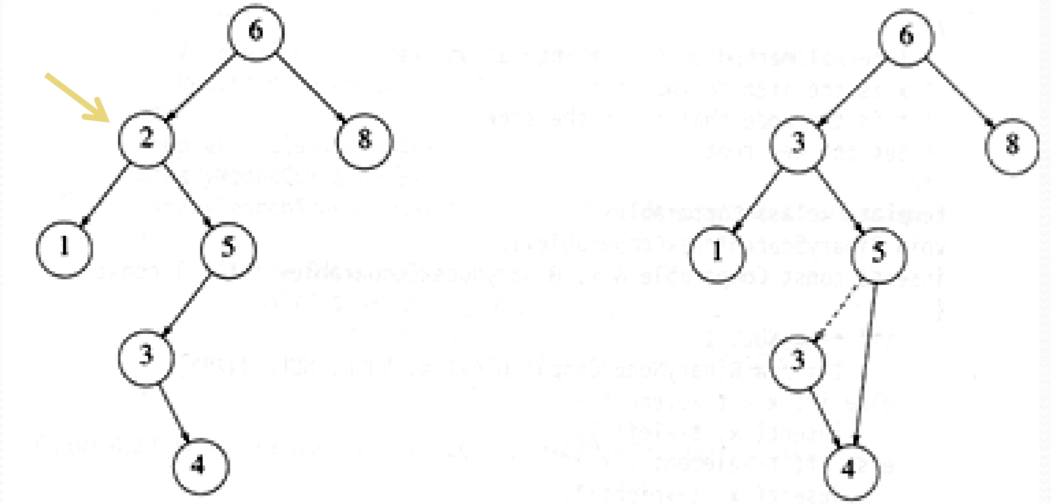
- Case I: Node to remove is a leaf node
  - Find it and set to null
- Case II: Node to remove has one child
  - Create new link from parent to child, and set node to be removed to null



Deletions in a BST - continued

Algorithm to remove Integer x

- Case III: Node to remove has two children
  - Find smallest node in right subtree – say it stores an Integer y. This node has at most one child
  - Replace x with y in node to be removed. Delete node that used to store y – this is done as in one of the two cases above



Average Case Analysis of BST Operations

- What does "average-case" mean in the case of BSTs? There is more than one interpretation, though in each case the result is the same. A BST with *n* nodes is said to be *randomly built* if *n* distinct integers are randomly chosen and inserted successively into an initially empty BST.
- The main result is:

  The average depth of a node in a randomly built BST having n nodes is O(log *n*).

- Therefore, the operations of insert, delete and search perform in O(log n) on average.
- It can be shown that in-order-traversal performs in O(n) on average. (next slide)

Optional: Running Time of In-Order Traversal

- **Algorithm:** InOrderTraversal(*node*)

  Input: A node *node* of a binary tree B with *n* nodes and with left child *left* and right child *right*

  Output: Every node in the subtree of B at *node* is marked/printed

```
if node ≠ null then
    InOrderTraversal(left)
    print(node)
    InOrderTraversal(right)
```

Optional: Running Time of In-Order Traversal – Guessing Method

- Let c be the time to evaluate "*node* ≠ null" and let d be the time spent within each self-call, other than other self-calls.

  Then T(0) = c (in this case the root is null). If B has a non-null root and left subtree has k elements then right subtree has n – k – 1 elements, then running time is given by:

  $$T(n) = T(k) + T(n - k - 1) + d$$

  We try values to guess a formula for T(n):

  $T(0) = c$
  $T(1) = T(0) + T(0) + d = c + d + c$
  $T(2) = T(1) + T(0) + d = (c+d+c) + c + d = (c + d) * 2 + c$
  $T(n) = (c+d) * n + c$

- **Verification**

  Let f(n) = (c + d)n + c. Clearly f(0) = c. We must show f satisfies the recurrence. Assume that it does for all m < n. We note that

  $$(c+d)n = (c+d)k + (c+d)(n-k-1) + (c+d).$$

  It follows that for any k for which 0 ≤ k ≤ n,

  $$\begin{aligned} f(n) &= (c + d)n + c \\ &= [(c + d)k + c] + [(c + d)(n - k - 1) + c] + d \\ &= f(k) + f(n - k - 1) + d \end{aligned}$$

Handling Duplicates in a BST

- Running times of find and insert given above, and their implementations, make the assumption that there are no duplicate values.

- To handle duplicate values, store in each Node a List (instead of a value); then when a value is added to a Node, it is instead added to the List.

- Example: Suppose we are storing Employees in a BST, ordered by Name. Our BST will be created so that the value in each node is a List<Employee> instance. Then, after insertions, all Employees with the same name will be found in a single List located in a single Node.

Using BSTs for Sorting

- Consider the following procedure for sorting a list of Integers:
  - Insert them into a BST
  - Print the results (or modify "print" so that it puts values in a list)

- *Exercise:* How good is this new sorting algorithm?

BSTs in the Java Libraries

- Java uses a special kind of BST (a "balanced" tree) as a background data structure for several of their data structures – namely `TreeSet` and `TreeMap`.

- If a BST becomes unbalanced, its performance degrades dramatically; techniques have been developed to keep a tree from slipping into an unbalanced condition – the most popular such technique produces *red-black trees.* `TreeSet` and `TreeMap` are implemented using red-black trees.

Introduction to Balanced BSTs: AVL Trees

- In order to avoid worst case scenarios for BSTs, several kinds of *balanced* BSTs have been devised. Typically, these work by formulating a *balance condition* ensures that, as long as the condition is satisfied, the BST (having n nodes) must have height O(log n). The balance condition is forced to be true by performing balancing steps after every insertion and deletion.
- One of the first kinds of balanced BSTs was the *AVL Tree* (A.V. L. are the first letters of the last names of the three guys who invented it).
- *The AVL Balance Condition:* For any node *x*, the difference between the height of the left subtree at *x* and the right subtree at *x* is at most 1. (We adopt the convention that the height of a tree having 0 nodes– i.e. a null root -- is –1)

Introduction to Balanced BSTs: AVL Trees

- **AVL Tree Height Theorem.** Every AVL tree having n nodes has height O(log n). Therefore, insertion, deletions, and searches all run in O(log n), even in the worst case.
- The proof is based entirely on the AVL Balance Condition. Using this condition, one shows by induction on height that if an AVL tree has height h and has n nodes, then $n \geq F_{h+1}$, where F is the Fibonacci sequence. Since F grows exponentially, it follows from this inequality that h is O(log n). (See the lecture note for this proof.)

Introduction to Balanced BSTs: Red-Black Trees

- Red-black trees are a more recent (and more efficient) alternative to AVL trees, though the balance condition is slightly more complicated.

- A BST is *red-black* if it has the following 4 properties:
  - Every node is colored either red or black
  - The root is colored black.
  - If a node is red, its children are black.
  - For each node *n*, every path from *n* to a NULL reference has the same number of black nodes.

Main Point

AVL trees are binary search trees in which remain balanced after insertions and deletions by preserving the AVL *balance condition.* The balance condition is: *For every node in the tree, the height of the left and right subtrees can differ by at most 1.* The balance condition is maintained, after insertions and deletions, by strategic use of *single* and *double rotations*. Worst-case running time for *insert, remove, find* is O(log *n*). The balance condition illustrates the principle that boundaries can serve to *give expression to* boundless intelligence rather than simply *limiting* that intelligence.

Connecting the Parts of Knowledge With The Wholeness of Knowledge

1. A Binary Search Tree can be used to maintain data in sorted order more efficiently than is possible using any kind of list. Average case running time for insertions and searchers is O(log n).

2. In a Binary Search Tree that does not incorporate procedures to maintain balance, insertions, deletions and searches all have a worst-case running time of Ω(n). By incorporating balance conditions, the worst case can be improved to O(log n).

3. *Transcendental Consciousness* is the field of perfect balance. All differences have Transcendental Consciousness as their common source.

4. *Impulses Within The Transcendental Field.* The sequential unfoldment that occurs within pure consciousness and that lies at the basis of creation proceeds in such a way that each new expression remains fully connected to its source. In this way, the balance between the competing emerging forces is maintained.

5. *Wholeness Moving Within Itself.* In Unity Consciousness, balance between inner and outer has reached such a state of completion that the two are recognized as alternative viewpoints of a single unified wholeness.